# Scheduling in distributed systems : survey and questions

Yasmina Belhamissi, Maurice Jégado

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1478

*Programme 3*
*Intelligence artificielle, Systèmes cognitifs et*
*Interaction homme-machine*

# SCHEDULING IN DISTRIBUTED SYSTEMS : SURVEY AND QUESTIONS

**Yasmina BELHAMISSI**
**Maurice JÉGADO**

**Juillet 1991**

*RR-1478*

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

# Scheduling in Distributed Systems: Survey and Questions

# Ordonnancement dans les Systèmes Distribués : Synthèse et Questions

Yasmina Belhamissi  Maurice Jégado
IRISA/INRIA
35 042 Rennes cédex, France

**Publication Interne n°592 - Juin 1991 - 36 pages - Programme 3**

**Abstract**

This report surveys various *scheduling* approaches that have been proposed in distributed computing systems. In a first step, we review the principles for *mapping* the parallel units of a single application to the processors of a distributed architecture. We then describe the main techniques for processor allocation used in distributed operating systems. In a second step, we summarize some scheduling proposals introduced in language oriented machines. This bibliography study raises some questions which lead us to the notion of a virtual machine whose aim would be to take into account both static and dynamic characteristics of an application and to enable the sharing of the physical architecture between various applications.

**Key words.** Scheduling, distributed architecture and system, parallel unit, mapping, processor allocation, virtual machine.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1  I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

**Résumé**

L'objectif de ce rapport est d'effectuer une synthèse de diverses approches proposées pour *l'ordonnancement* dans les systèmes informatique distribués. Dans un premier temps, nous étudions les principes du *placement* des unités parallèles d'une application (unique) sur les processeurs d'une architecture distribuée. Nous décrivons ensuite les principales techniques d'allocation de processeurs, employées dans les systèmes opératoires distribués. Dans un second temps, nous résumons quelques propositions d'ordonnancement introduites dans des machines orientées langage. Cette étude bibliographique pose plusieurs questions qui nous conduisent à la notion de *machine virtuelle* dont le but serait de tenir compte des propriétés statiques et dynamiques des applications et de permettre le multiplexage de l'architecture physique entre diverses applications.

**Mots clés**. Ordonnancement, architecture et sytème distribué, unité parallèle, placement, allocation de processeurs, machine virtuelle.

# Contents

# 1 Introduction

Historically, parallelism has been first studied by the operating system designers. Today, parallel programming is applied to numerous application domains. One of the main motivations is to exploit the potential parallelism offered by a distributed architecture (comprising multiple processors) so as to improve the execution efficiency of the applications [Banâtre 90].

The objective of this report is to survey various *scheduling* approaches that have been proposed in distributed computing systems. Scheduling in distributed systems is a difficult subject; numerous works - depending on different application domains (real time) and particular hypothesis (architecture type) - have been published. This report is organized in the following way.

In section 2, we introduce a basic formulation of the scheduling problem. The *mapping* of the parallel units of an application to the architecture processors may use mathematical models based on graph theory and objective functions. When both the application characteristics and architecture features are statically defined, it is natural to search for an efficient static mapping of the tasks (parallel units) onto the architecture processors. This is examined in section 3. In other cases, static methods must be revised in order to take into account the dynamic aspects of the computation. This is examined in section 4.

In section 5, we deal with general purpose operating systems. We first briefly recall scheduling techniques employed in centralized operating systems and then examine the main scheduling issues in distributed operating systems. An operating system provides the support mechanisms for implementing various scheduling policies which may be user-defined or system-defined by default. In the latter case, the scheduling techniques applied can be understood as the application of some of the basic principles introduced previously to a particular application context.

In section 6, we consider more abstract models of computation and examine scheduling issues within *language oriented* machines. Two examples pertaining respectively to *explicit* and *implicit* parallelism language classes are dealt with and reveal that the model of computation may have a deep influence on the scheduling issues.

Static and dynamic scheduling of an application, scheduling in operating systems are subjects which are generally treated separately by the authors. We are examining them voluntarily in this same report since we believe that each of them raises relevant questions to scheduling.

Static scheduling algorithms incur a low overhead during execution and have the merit to define a rigorous framework for performance evaluation. However, they are limited to applications which characteristics must be exhibited statically (for instance, dynamic process creation is not considered); they do not take into account either the sharing of the architecture (machines) between various applications. Dynamic scheduling algorithms relax the constraint that application characteristics must be exhibited statically at the price of a higher overhead during

execution but still do not address the sharing problem. General purpose operating systems do address the sharing of the machine between several applications but, given the diversity of the applications submitted to execution, make little use of the application characteristics in order to optimize the performances.

Would it be possible to define a scheme which would take the benefits of each of these approaches without the drawbacks? This question leads us to the notion of a virtual machine discussed in section 7. A main conclusion of this work is that the investigation of scheduling issues within the framework of a language oriented machine appears less difficult than tackling the problem in full generality.

# 2 Scheduling a parallel application: principles

In this section, we present the basic principles for scheduling a parallel application. First, some definitions are introduced. Second, the scheduling problem is stated and finally, basic scheduling approaches are discussed.

## 2.1 Basic definitions

In [Bal 89], two types of distribution namely *logical* and *physical* are clearly identified. Logical distribution refer to software processes communicating by explicit message passing as opposed to logically nondistributed software in which software processes communicate through shared data. Physical distribution refer to architectures which do not have shared memory, this may be contrasted to multiprocessors which have a single systemwide primary memory. There are four combinations of logical and physical distribution: (1) logically distributed software running on physically distributed hardware, (2) logically distributed software running on physically nondistributed hardware, (3) logically nondistributed software running on physically distributed hardware, and (4) logically nondistributed software running on physically nondistributed hardware.

In this report, we are primarily interested in the first of these combinations namely logically distributed software running on physically distributed hardware. In order to give a precise definition to the scheduling problem, a basic model of computation together with a simple architecture model are introduced below. Those models are not claimed to cover all of the various scheduling proposals that have been published nor realistic in all situations. However, they have the merit to be a sound basis for most of the works reviewed. Limitations and extensions of these models are discussed in the text at various points.

We adopt the most basic model of computation: a parallel application is supposed to comprise a concurrent group of sequential processes communicating through message passing. The words task and process are used interchangeably in the text. More formally, a parallel application $A$

is defined as a pair $A = (< T, C >, time)$ described below.

- $< T, C >$ is an undirected graph representing the communicating tasks of the parallel application:
  - T=$\{t_1, t_2, ..., t_n\}$ is a set of tasks ($n$ denotes the number of tasks),
  - $C$ is the adjacency cost matrix of the communication patterns such that $c_{ij}$ denotes the number of information units transmitted from task $t_i$ to task $t_j$; the *communication intensity* between $t_i$ and $t_j$ is defined as $c_{ij} + c_{ji}$.
- *time* is a function from the task set $T$ onto the set of real numbers such that $time(t_i)$ denotes the number of atomic computational steps (instructions) performed by the task $t_i$. Rather than the time function of a task $t_i$, we will often use the notion of the execution time of the task (assuming that all processors have identical speed, the execution time of a process $t_i$ may be approximated, for instance, multiplying the average instruction execution time by $time(t_i)$).

At this stage, we do not detail how the previous characteristics could be obtained. Note also that in the previous definition, processes are not linked by *precedence constraints*. In other words, processes can be executed in an arbitrary order.

A process is a sequential unit which can be executed on a processor of the distributed architecture. The distributed architecture is defined by an undirected graph $H =< P, L >$ described below.

- $P$=$\{p_1, p_2, ..., p_p\}$ is the set of processors ($p$ denotes the number of processors).
- $L$ is the adjacency cost matrix of the architecture graph, $l_{wq}$ denotes the costs of transferring one information unit between processors $p_w$ and $p_q$.
- We assume on the one hand, that processors are homogeneous and do not share memory and on the other hand, that $H$ is a *connected* graph i.e. there exists a path (direct or indirect) between each pair of processors belonging to $P$.

## 2.2 The scheduling problem

One of the main reasons for executing an application on a distributed architecture is to improve the execution efficiency through the use of multiple processors that run in parallel [Banâtre 90].

For instance, consider the following simple example where $A = (< T, C >, time)$ is a parallel application and $T$ =$\{t_1, t_2\}$. We assume that the tasks are independent (do not communicate) i.e. $(\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : c_{ij} = 0)$. The *time* function is arbitrary. The parallel application $A$ is executed on an architecture $G = (P, L)$ where $P$ =$\{p_1, p_2\}$, and $L$ is arbitrary. If both tasks were assigned to a same processor, they would be executed in pseudo-parallelism

leading to a response time equal to $time(t_1) + time(t_2)$. A parallel execution on both processors will lead to a lower response time equal to the maximum of $time(t_1)$ and $time(t_2)$.

The scheduling problem consists of two subproblems. First, a *mapping* function $M$ of the application tasks on the architecture processors such that

- $M : T \text{—>} P$, and
- $M(t_i) = p_r$ meaning that $t_i$ is assigned to processor $p_r$,

has to be computed. Second, the process(es) mapped on a given processor must be *locally* scheduled. It should be noted that the term scheduling is used by some authors to refer to some correct execution order that must be enforced so as to satisfy synchronisation constraints. In this report, scheduling is envisaged according to a performance point of view. The main efficiency criteria considered is the *overall response time* of the parallel application i.e. the interval between the request for execution and the execution completion.

At this stage, we do not consider the possible sharing of the architecture between several concurrent applications and we only examine the scheduling of a *single* application. Sharing will be considered later in section 5. Many references do not address properly this issue; in our view, separation of these concerns helps making the scheduling problem much more tractable. Another point worth stressing is that in the basic formulation of the scheduling problem, we are not concerned with the placement of the *images* (ex. data) of the application tasks into the architecture memories (not modelled above) but only with the scheduling of the tasks onto the processors. In other words, at this stage, one may suppose that each processor is equipped with a private memory large enough for containing all the images of the processes mapped to it.

It is difficult to define precisely all the factors which may influence the response time of an application. However, there is a consensus in the literature that the response time depends at least on two fundamental factors which are summarized in Fig. 1. In the following, we discuss each of them illustrated by examples.

1. Processors' load
2. Communication costs between processes

Figure 1: Scheduling factors of a parallel application

**Processors' load.** A possible estimate of the load of a processor $q$ would be the following:

$load(q) = \sum_{i:1 \leq i \leq n, (M(t_i)=q)} time(t_i)$.

Obviously, a process being executed on an overloaded machine will need more time to perform its computation than if it were executed on a non-loaded one. Consequently, the processors' load

has an influence on the response time. A sensible optimization action would be to balance the load between the processors i.e. $(\forall i : 1 \leq i \leq p : load(q_i) \simeq \frac{\sum_{j:1\leq j\leq p} load(q_j)}{p})$. To illustrate this, let add a complementary task $t_3$ to the application $A$ (introduced previously) and define the function *time* as: $time(t_1) = 5$, $time(t_2) = 5$, and $time(t_3) = 10$. Only two mappings among the eight ($2^3$) distinct possible mappings balance exactly the processors' load, namely, the mappings $M_1$ ($M_1(t_1) = p_1$, $M_1(t_2) = p_1$, and $M_1(t_3) = p_2$), and $M_2$ ($M_2(t_1) = p_2$, $M_2(t_2) = p_2$, and $M_2(t_3) = p_1$). The response time of each of these mappings is then equal to $(max_{i:1\leq i\leq p} load(p_i)) = 10$ while in the other cases it would be greater, and equal to 20 (5+5+10) in the worst case.

**Interprocess communication cost.** As stated previously, we are not concerned with scheduling as a means for satisfying synchronisation constraints between processes. But, it is clear that in practice, processes are rarely independent and are in relation with each other. Synchronization constraints may *delay* the processes at the synchronization points of their computation and in this sense they are of concern to us since they may influence the overall response time of the application. To this end, we take into account the communication patterns of the processes in the basic scheduling model.

In the literature, we find basically two strategies for mapping communicating processes on a distributed architecture [Tanenbaum 85]. The first one is to assign processes which communicate together on different processors, so that they can run in parallel rather than in pseudo parallelism. The second one is precisely the opposite, namely, to assign communicating processes to the same processor. How and under which conditions can these strategies reduce the response time of an application? In our view, a main criteria for responding to this question is the time needed by the communication sub-system for sending a message. For illustration purposes, let come back to the first example introduced at the beginning of the section, assuming now that the processes $t_1$ and $t_2$ are communicating processes ($c_{12} \neq 0$) ; recall that $M(t_1) = p_1$ and $M(t_2) = p_2$.

Let assume first (as depicted in (a)) that the communication is fast and does not induce any waiting delay on the recipient task $t_2$; the *send* operation is assumed asynchronous. The response time is then as before equal to $max(time(t_1), time(t_2))$. Clearly, in this case, the first strategy (mapping $t_1$ and $t_2$ on separate processors) compares favourably to the second one (mapping $t_1$ and $t_2$ on the same processor) which would lead to a response time equal to $time(t_1) + time(t_2)$.

Assume now that the transfer delay is very long compared to the execution times so that $t_2$ is delayed waiting for a message as depicted in (b). The response time of the application is then equal to $(t_{11} + \triangle_b + t_{22})$ which is greater than $(time(t_1) + time(t_2)) = t_{11} + t_{12} + t_{21} + t_{22}$. In this case, the first strategy behaves poorly compared to the other one.

In a multiprocessor architecture equipped with a systemwide shared memory, interprocessor communication is likely to be fast and would suggest assigning highly communicating processes

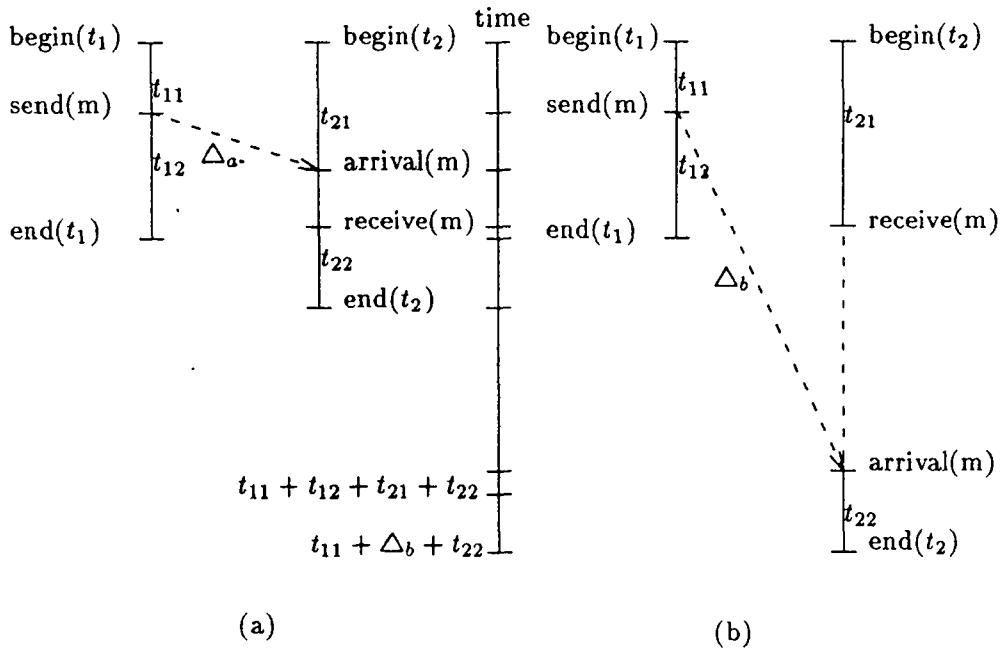


Figure 2: Influence of the communication delays on the overall response time

on separate processors. However, recall that we are assuming a distributed architecture and therefore we assume, in the remainder of the text, that highly communicating processes should be, as much as possible, clustered on a same processor so as to avoid that communication delays impinge on the response time of the application. A rough way to achieve this goal is to minimize the *communication costs* of a mapping defined as $\sum_{i,j:1\leq i\leq n,1\leq j\leq n} c_{ij} * l_{M(t_i),M(t_j)}$. Notice that the term costs must be understood here as a penalty on the response time and not as the price of the communications. Given the previous expression, a simple way to favour intra-processor communication would be to set the diagonal of the matrix $L$ to zero but more refined choices are possible.

The reader may question whether handling communication issues as above is not too much restrictive. A more precise communication model would take into account the *precedence constraints* induced by the message delivery. For instance, in the previous example, the subunit $t_{22}$ may only start when the subunits $t_{11}$ and $t_{21}$ have finished and the message $m$ has been delivered. However, most scheduling algorithms estimate the communications costs in a manner similar to above. A more precise model but limited to specific situations can be found for instance in [Anger 90].

**The dependency relation between the scheduling factors.** Following the previous discussion, it might appear that balancing the load together with a minimization of the communication costs are sufficient to obtain an efficient mapping. But unfortunately, this strategy is

inconsistent because these scheduling factors are not independent of each other as we illustrate it now.

When scattering the concurrent processes on separate processors so as to balance the load between the processors, two cases have to be considered. If the concurrent processes do not communicate intensively, then the overall response time will be better. However, if communication intensity is high, the overall response time will suffer, at the extreme this may even cancel the improvement due to the parallel execution (as shown in case (b) of the previous example). Now, as stated above, if we want to prevent that the communication costs impinge on the overall response time, it is preferable to gather intensively communicating processes on the same processor. But, this may lead to situations where some processors are overloaded compared to others i.e. the load is not balanced. Consequently, designing an efficient scheduling algorithm amounts to finding a judicious compromise between both previous factors.

## 2.3 Scheduling approaches

The mapping of concurrent processes onto processors is typically an *optimization problem* and in this sense is similar to many other optimization problems encountered in resource management such as allocating the communication channels of a communication network (routing) or placing data on a disk for example. However, it should be noted that the mapping problem has been shown *NP-complete* in all but very restricted cases; such results can be found in [Ullman 75] and elsewhere. The complexity of this problem is bigger yet when dealing with real-time applications where some deadline constraints must be taken into account. To simplify this presentation, mapping algorithms for real-time applications are not examined; the interested reader may refer to [Ramanritham 89] for example. Generally speaking, the exact solution to NP-complete problems is difficult to compute in practice since this would be too much resource demanding. Rather, heuristics delivering suboptimal solutions which require acceptable resources are computed.

Broadly speaking, scheduling has been tackled from two points of view: static and dynamic. Relying on the structuring principle of a system in terms of *policies* and *mechanisms*, this can be defined as follows. *Static scheduling* refer to scheduling policies applied prior to execution while *dynamic scheduling* refer to scheduling policies applied during the execution. Static mapping is examined in section 3 while dynamic mapping is examined in section 4. Local scheduling of processes mapped on a processor and basic scheduling mechanisms are dealt with in section 5 where processor allocation in distributed operating systems is reviewed.

Static mapping might be preferred since this incurs a low overhead during the execution as opposed to dynamic mapping. However, although this is not always clear in the literature, static mapping may not be always feasible and dynamic mapping be necessary in some circumstances. This is the case if the application exhibits dynamic parallelism (as opposed to static parallelism) or if the architecture can be dynamically reconfigured. Let us define more precisely these notions.

- *Static parallelism.* This type of application excludes the possibility to dynamically create new processes; the patterns of communications between processes must also be known statically.

- *Dynamic parallelism.* These applications enable a dynamic evolution of the computation: processes may be created dynamically and the pattern of communication may change during the execution.

- *Dynamically reconfigurable architecture.* If the architecture is dynamically reconfigurable, the number of processors may change during the execution (processors may be added or removed); the interconnection topology may also change.

As reviewed in [Chu 80], two broad methods are used to model the mapping problem:

- Given the basic definitions introduced in section 2.1, the mapping problem can be considered as a graph partitioning problem. The aim of the partitioning is to assign each partition of the application graph to a processor (the partitioning must of course optimize the scheduling factors introduced previously).

- The second method relies on the definition of a so-called *objective function* which takes as arguments various parameters modelling the mapping problem. The optimization of this function through appropriate mathematical programming techniques determines the mapping.

Finally, two types of assignment which we call *invariable assignment* and *variable assignment* are to be distinguished. When invariable assignment is used, a process is assigned to the same processor during the entire execution. When variable assignment is preferred, a process may *migrate* during its execution.

Our aim in the following sections 3 and 4 is to introduce a set of basic tools and methods useful to define various scheduling policies. The objective is not to compare these methods whose applicability may depend on the type of application. For instance, the parallelism *grain* may have a deep influence on the kind of scheduling policy which may be used.

# 3 Static mapping algorithms

Algorithms for static mapping require the descriptive costs of the application characteristics and the architecture features to be estimated statically prior to execution. The definitions adopted in this section are those of the basic model introduced in section 2.1 except for *time*. In order to take into account execution costs of a task on different processors, *time* is redefined as a matrix $TIME$ such that $time_{ir}$ denotes the execution time of the task $t_i$ on processor $p_r$.

The classification of the static mapping algorithms published in the literature is not an easy task, our aim is not to give an exhaustive presentation but to introduce some fundamental principles. Static mapping based on graph models is discussed in section 3.1 while objective functions are presented in section 3.2.

## 3.1 Graph modelling

The mapping problem can be tackled as a graph partitioning problem. The mapping of the $n$ tasks onto the $p$ processors results in an adequate dividing of the graph into $p$ partitions, each partition being mapped onto a processor. Interpartition links represent costly interprocessor communication while intrapartition links model intraprocessor communication whose cost is supposed to be negligible. For example, let us consider an application consisting of eight tasks; the communication intensity between each pair of tasks is assumed to be identical. Two different mappings are depicted in Fig. 3. As far as the minimization of the communication costs is concerned, the partitioning of Fig. 3.b is better than that of Fig. 3.a since the number of interpartition communication links is lower.

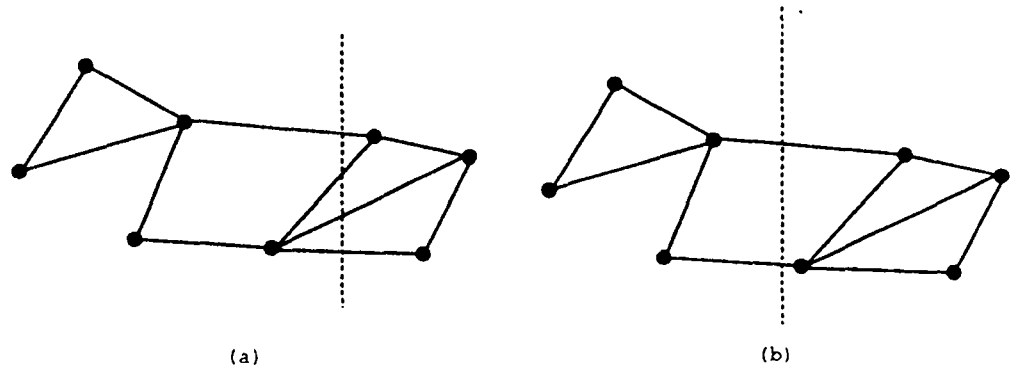


Figure 3: Examples of graph partitioning

To our knowledge, there does not exist any exact graph partitioning algorithm which resolves the mapping problem in full generality. In the particular case of a distributed system comprising only two processors, Stone [Stone 77] has adapted a maxflow/mincut algorithm of a graph to solve the mapping problem. This algorithm yields an optimal solution (regarding the response time) based on the optimization of both the communication costs and the execution time requirements of the tasks. Unfortunately, this algorithm cannot be used beyond two processors.

Lo [Lo 85] developed a general mapping algorithm which assumes the same model as introduced in section 2.1. However, the per-unit cost of information transfer between all pairs of processors is assumed to be the same. The parallel application graph and the architecture graph are merged together so as to obtain a graph thus containing two kinds of nodes (processor and task nodes). An intertask edge is valued by the intertask communication intensity while the value

of an edge linking a processor node $p_r$ to a task node $t_i$ denotes the execution time $time_{ir}$ which would be required for executing the task on the processor. The principle of this algorithm is to apply successively the maxflow/mincut algorithm in order to obtain a final partitioning of the graph such that each partition contains exactly one processor (tasks belonging to a same partition are mapped on the processor node of the partition). A partitioning example is depicted in Fig. 4. The parallel application $A = (< T, C >, TIME)$ is such that $T = \{t_1, t_2, t_3, t_4, t_5\}$; precise definition of $C$ and $TIME$ would be easily deduced from the values given in Fig. 4. The application $A$ is assumed to be executed on a parallel architecture $G = (P, L)$ where $P = \{p_1, p_2, p_3\}$ and $L$ is arbitrary. The partitioning depicted in Fig. 4 defines a mapping $M$ such that $M(t_1) = p_1$, $M(t_2) = p_1$, $M(t_3) = p_3$,$M(t_4) = p_2$, and $M(t_5) = p_3$.

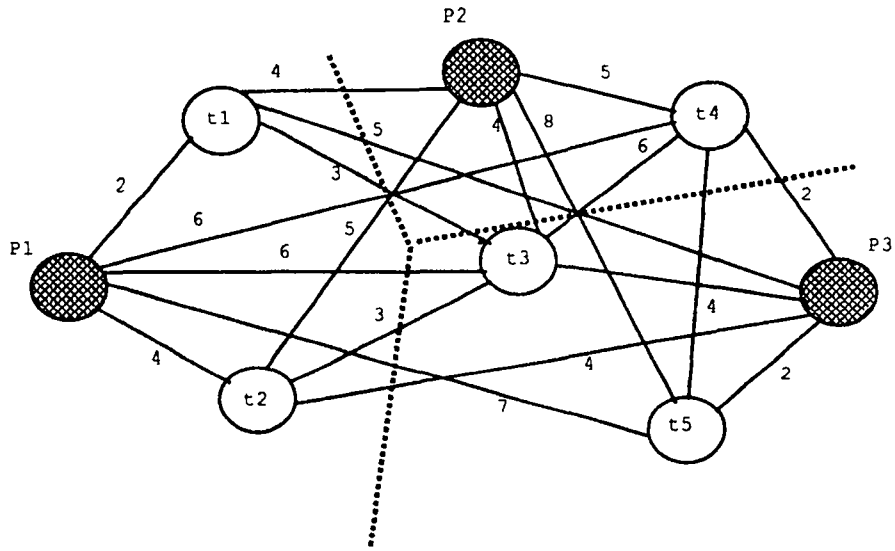


Figure 4: Graph partitioning: example of Lo's algorithm

In [Lo 87], Lo developed a second algorithm based on the same assumptions as stated above. However, this algorithm takes into account a complementary notion to the basic model introduced in section 2.1, namely the *interference costs* between tasks. For instance, two tasks which are both compute-bound have a high interference cost since it would not be judicious to assign them to a same processor (they would execute in pseudo-parallelism). On the contrary, a compute-bound task and an i/o-bound task have a low interference cost since their executions could overlap on a processor. Interference costs could be conveniently modelled by a matrix $S$ such that $s_{ij}$ denotes the incompatibility of tasks $t_i$ and $t_j$ to be assigned onto a same processor. Notice that interference costs optimization may conflict with communication costs optimization. For instance, let us consider a pair of high interference and highly communicating tasks; the optimization of the former factor tends to scatter the tasks on distinct processors while the optimization of the latter tends to gather both tasks on the same processor. In [Lo 87], a solution

minimizing both the costs discussed previously and interference costs is given.

Kim [Kim 88] also applies a graph technique to resolve the mapping problem of the tasks onto the processors. The parallel application is modelled by a *directed* graph whose edges denote *dependency relations* between communicating tasks. We understand that the notion of dependency relation should not be confused with precedence constraint: if $t_1$ sends information to $t_2$, a directed edge between $t_1$ and $t_2$ is recorded denoting the dependency relation between both tasks but this does not mean that $t_2$ may only start when $t_1$ has finished. The architecture is modelled as in section 2.1. The assignment of tasks to processors is obtained by a mapping of the computation graph onto the architecture graph. Process nodes of the application graph which are linked by dependency relations may be merged together before the final mapping onto the architecture graph is performed.

In summary, graph modelling is an adequate method for the static mapping of a parallel application. In our view, a drawback of this approach is that the load balancing factor (Fig. 1) is neglected to the advantage of the communication costs factor. Modelling by objective functions, to be detailed now, allows easier parameterization.

## 3.2 Modelling by objective functions

The role of a so-called objective function (or cost function) is to take into account the various parameters that can influence an objective. Reaching the objective requires the minimization of the cost function possibly under some constraints through appropriate mathematical techniques.

In the following, we first introduce an elementary cost function and progressively add complementary costs thereby refining the specifications of the mapping problem. This style of presentation is borrowed from [Andre 88]. In all the examples, we assume that intraprocessor communication cost is negligible and use the same notations as before. In contrast to the previous section where we discussed the basic principles for resolving the mapping problem when a graph approach is used, we state below some illustrative cost functions of the mapping problem but do not explain the principles of the mathematical techniques that may be used for the minimization. The interested reader is referred to [Manber 89] for a synthesis of such methods.

**Elementary cost function of communications.** Assuming that the cost of each interprocessor communication is the same, the function $f_1$ below is the function to be minimized in order to optimize the communication costs of a parallel application. Let

$f_1 = \sum_{i,j:1\leq i\leq n, 1\leq j\leq n, i\neq j, M(t_i)\neq M(t_j)} c_{ij}$ where

$c_{ij}$ = communication intensity between tasks $i$ and $j$.

Clearly, the minimization of $f_1$ would lead to the mapping of all tasks on a same processor. But, as stated above, it is generally the case that the optimization has to be performed under some constraints. For example, the constraint $(\forall r : 1 \leq r \leq p : (Ni : 1 \leq i \leq n : M(t_i) = r) \leq k)$

where $N$ denotes the counting quantifier expresses that at most $k$ tasks may be assigned to a same processor.

**Cost function of communications over an architecture with diverse links.** The function $f_2$ below is the function to be minimized to optimize the communication costs of a parallel application when an architecture with diverse links is used. Let

$f_2 = \sum_{r,q:1\leq r\leq p,1\leq q\leq p,r\neq q} \sum_{i,j:1\leq i\leq n,1\leq j\leq n,i\neq j,M(t_i)=r,M(t_j)=q} c_{ij} * l_{rq}$ where

$l_{rq}$ = per-unit communication cost between processors $r$ and $q$, and

$c_{ij}$ = communication intensity between tasks $i$ and $j$.

Again, without further constraints, the optimization of $f_2$ would lead to the obvious mapping of all the tasks onto a same processor.

**Cost function of communications and execution times.** Assuming that processors are of different speed and that the architecture consists of diverse links as before, the cost function $f_3$ below takes into account both communication and execution aspects where $e$ denotes the execution costs. Let

$e = Maximum_{r:1\leq r\leq p}(\sum_{i:1\leq i\leq n,M(t_i)=p_r} time_{ir})$ where

$time_{ir}$ = execution time of task $t_i$ on processor $p_r$; and

$f_3 = f_2 + e$.

The optimization of $f_2$ tends to cluster the tasks on a same processor while optimization of $e$ tends to the contrary namely scattering processes on different processors. The expression $f_3$ captures these conflicting aspects in a simple manner, the costs of each being simply added.

**Cost function taking into account interference costs.** Recall that Lo [Lo 87] has introduced the notion of *interference cost* for modelling the incompatibility of two tasks to be assigned to a same processor. The function $f_4$ below takes into account this parameter; $s$ denoting the interference costs.

Let

$s = \sum_{l:1\leq l\leq p} \sum_{i,j:1\leq i\leq n,1\leq j\leq n,i\neq j,M(t_i)=M(t_j)=l} s_{ij}$ where

$s_{ij}$ = interference cost between tasks $i$ and $j$; and

$f_4 = f_3 + s$.

**Cost function with load balancing.** The function $f_3$ takes into account execution parameters in a rough way, the function $f_5$ below takes into account the load balancing factor in a more precise manner. (The scheduling algorithm of [Ryou 88] integrates the load balancing factor in a manner similar to the function $f_5$ below although this algorithm may also accomodate some form of sharing as discussed further in this report). Let

$f_5 = f_2 + s + \sum_{r,q:1\leq r\leq p,1\leq q\leq p,r\neq q}(L_r - L_q)$ where

$L_r$ = load of processor $r$, and
$L_q$ = load of processor $q$.

The purpose of the above examples was to illustrate the use of the cost functions for specifying the mapping problem. In practice, the definition of an optimum cost function may induce more complexity than above, attributing for instance a weight to each parameter. A main advantage of expressing the mapping problem by a cost function, compared to, say, the graph approach is extensibility allowing to take into account numerous criterion in the problem specifications. Therefore, the expression power of cost functions would appear bigger, and in some sense, the graph algorithms discussed in the previous section can be understood as resolution methods of specific cost functions.

## 3.3 Comments

Static mapping algorithms require as inputs the costs which describe the application behaviour and the architecture features. As far as the application is concerned, one must notice that the descriptive costs represent a *static approximation* of the dynamic behaviour of the application. In practice, these might be difficult to determine. For instance, the execution time of a task could possibly be estimated by a static analysis of the task (the reference [Chu 80] for instance describes some methods for that purpose) or alternatively by running the task once before this figure be used by a mapping algorithm. But, the exact execution time is very often dependent on the input data which may be not known prior to a particular execution. The communication costs may depend on the data distribution which again may be unknown prior to execution.

As far as the architecture is concerned, static mapping algorithms assume a non dynamically reconfigurable hardware and generally suppose that the network is fully connected. However, one can notice that if there does not exist a communication path between two processors, the communication cost between these processors can be conveniently set to the infinite value. Many static mapping algorithms have been developed for particular kinds of architecture. For example, in [Bokhari 81], an algorithm tending to map communicating processes onto neighbouring processors of a processor array machine is described. The hypercube machine has given rise to numerous studies [Sadayappan 90, Ercal 90]. In [Chrétienne 91], ring networks are examined. Most of these algorithms attempt to find a mapping of the communicating processes as close as possible to the network topology so as to avoid data routing. Concerning the characteristics of the communication channels, Colin [Colin 89] has proposed a mapping algorithm taking into account the possibly limited capacity of the channels.

An issue which we have not considered is the stability of the previous algorithms that is their

sensitivity to input changes. In [Ji 90] where a multiprocessor context is assumed, it is claimed that slight changes of the input values would have little effects on the execution efficiency of the resulting mapping. Although we have not deepened this subject, it would appear that this is also the case when a distributed system is considered.

As stated in section 2, we are primarily interested in scheduling logically distributed software on physically distributed hardware. Some authors have investigated the scheduling problem in other contexts. For instance, a model of computation based on shared objects is assumed in [Schwan 85]. The static mapping algorithm proceeds in two steps. In a first step, objects are clustered with the tasks which use them intensively. In a second step, tasks and objects are mapped onto the architecture sites.

# 4 Dynamic mapping algorithms

In contrast to the static mapping methods, dynamic methods do not impose that the future behaviour of the application be known. Most of dynamic methods model the scheduling problem by an objective function which is optimized several times during the execution as to adapt the mapping to the changing behaviour of the application or accomodate possible architecture reconfigurations. In the literature, many dynamic scheduling algorithms are often called load-balancing algorithms. However, this term may be somewhat misleading since, as explained in section 2, performances are dependent on processors' load but also of the interprocess communication costs. In summary, a dynamic scheduling algorithm should ideally optimize at least both factors of Fig. 1.

Two main questions are to be answered by a dynamic algorithm.

- Where (on which processor) should a newly created process be assigned? Two choices are possible. First, the process may be assigned onto the *origin site* on which the creation request is processed. Second, an appropriate machine optimizing the application performances may be selected; the process then starts its execution on that machine.
- How to manage the mapping modifications (migration) during the application execution? Migration is constituted of two parts.
  - *Decision.* The role of a decision policy is to answer to the following subquestions: (i) when should the mapping be reconsidered? (ii) which processes should move from their current site? (iii) where (on which site) should a moving process be transferred?
  - *Transfer.* The role of a transfer mechanism is to move the execution environment of a process from its *current* machine to a new residence site.

In this section, we are not concerned with the transfer mechanism. This is an operating system dependent function and is examined in section 5.

Many algorithms have been proposed for answering to the previous questions. An exhaustive classification would be a difficult task. Some authors classify them into two broad categories: receiver initiated and sender initiated. An algorithm is said receiver initiated if the mapping reconsideration is initiated by a machine willing to receive one or more processes while it is said sender initiated if the mapping reconsideration is initiated by a machine willing to export one or more processes. In the following, we introduce three classes of algorithms according to an increasing complexity order. First, a simple method, namely demand load balancing, which is receiver initiated is discussed. Second, bidding like methods which are sender initiated are presented. The example of Stankovic's algorithm based on this principle is then described. Lastly, we present a more abstract method based on a physical modelling of the scheduling problem.

## 4.1 Demand load balancing

Among the various factors that may impinge on the application performances, load considerations are essential. *Demand load balancing* is a simple strategy which aims at avoiding *inactive* (or idle) sites.

Within this framework, the most natural way to assign a newly created process is to assign the process to the local machine on which it is created. Load examination can be performed at various stages in order to organize migrations which would balance the load between the processors. However, demand load balancing does consider possible migrations only when a site is becoming inactive (null load). An inactive site enquires after the other sites to get some work. Notice that this is a fully decentralized strategy. We do not describe in detail here how the selection of potential migrating candidates to be executed on the inactive site can be performed.

In summary, this strategy has the merit of simplicity but takes into account only the processors' load for optimizing performances. Yet, load considerations are dealt with in a rough manner.

## 4.2 Bidding methods

The basic principle of *bidding* is the following. When a process is a candidate for work, a request telling what it needs (ex. execution time) is broadcast to other sites. Sites can then bid for the work and answer to the caller which chooses the best answer among those which have been received. Bidding can be used both to answer to the initial assignment question and to manage migration during execution as discussed in the following.

For example, the basic steps for selecting an appropriate machine at initial assignment might be:

1. send requests containing the characteristics of the process to execute to the other machines;

2. receive the answers corresponding to these requests;

3. choose a site among those who answered positively and transfer the process onto this site; and

4. if no positive answer is received, execute the process on the local machine.

In order to manage the migration during execution, bidding can be used to determine the potential recipient machines which could be better alternatives for processes realizing poor performances on their current sites.

As an illustration of a migration algorithm based on the bidding principle, we introduce below a specific algorithm.

### The Stankovic's algorithm

In Stankovic's proposal [Stankovic 84], the bidding algorithm is replicated at each site. The same algorithm is used both to determine possible process candidates for migration as well as to respond to request for bids received from other sites.

The migration proceeds in two steps. In a first step, each site periodically evaluates all local processes by means of a so-called *decision function.* Processes whose computed value is greater than a predefined threshold are deemed to execute with good performances on the site while others are potential candidates to migration. In a second step, sites which could accept the candidates to migration are to be determined. In order to achieve this goal, the characteristics of candidate processes are sent to the other sites. Each site receiving a migration request evaluates the decision function for the candidate process and returns the computed value to the sender. The sender can then possibly move the process to the site which has returned the biggest value.

The decision function $F$ is an objective function which may take as input $n$ *stimuli ('excitations')* $E_i$ and $m$ *inhibitors* $I_j$. This function is used to model the scheduling parameters of a process. For instance, an inhibitor could be a dynamic parameter of a process such as: the process has already moved $x$ times (assuming we do not wish the process to move more). Yet another inhibitor could be a static characteristic of a process such as: the task uses a special resource which is available only on the given site. Stimuli are basically used to represent the costs associated to a scheduling strategy (notice that in this case, the optimization results in maximizing the decision function).

For example, let consider a simple strategy which aims at optimizing only the load factor i.e. balancing the load of the processors used by the application. For the sake of simplicity, we are assuming that the execution time of a process is statically known and that the system maintains the cpu-time consumed by a process so as to determine its remaining execution time. A simple decision function associated to a process would then entail a single stimulus $C$ such that: $C = 1$ / (Sum of the remaining execution times of the processes residing at the site + Execution time requirement of the process)

Many type of scheduling parameters could be handled by the decision function: the reference [Stankovic 84] outlines some illustrative examples. Another point worth stressing is that, the same decision function is used by a site both for responding to a bid request from another site and evaluating each local process so as to determine the migration candidates. However, in the latter case, the threshold value must be determined (as far as we understand, this issue has not been addressed in [Stankovic 84]). For instance, in the previous load-balancing strategy, the threshold value could be naively set to infinite but, the ideal threshold would be the average load value which, to be determined, would require some complementary work.

### 4.3 A dynamic method based on a physical model

Some authors have considered the dynamic scheduling problem in a more abstract way than stated above and modelled the dynamic mapping of the parallel tasks of an application by a *physical system of moving particles*. Tasks are modelled by particles, task migrations in the distributed architecture are modelled by particle moves in the physical system [Fox 88]. Searching for an optimal dynamic mapping of the tasks onto the processors results in minimizing an objective function called *energy function* using an *annealing* algorithm. Particle moves that minimize the energy function are all allowed whereas moves increasing the energy function are disallowed beyond a given probability threshold. The energy function depends on several forces which model the scheduling factors introduced in Fig. 1.

- Load balancing is modelled by a repulsion force between particles which tends to disperse the particles in the whole system in a balanced and homogeneous way.
- Intensively communicating tasks are modelled by an attractive force between the corresponding particles. This force is proportional to the quantity of information exchanged.
- Moving a task from a processor to another one is a costly operation and should be performed only if the benefit gained is substantial. In order to avoid low benefit moves, the physical model is completed by an attractive force of particles to their current location.

This concludes the presentation of dynamic scheduling methods.

## 5 Processor allocation in distributed operating systems

In this section we are concerned with processor allocation in distributed *operating* systems. We assume that the execution model offered by the operating system is similar to the model of the previous sections: a parallel program (application) is made up of multiple concurrent sequential processes which communicate by message passing. A process is an execution entity which is

assigned to a processor. Moreover, we are assuming that no *a priori static* information about the processes' characteristics are known.

An essential function of an operating system is to provide to users a *virtual machine* more abstract than the physical one [Krakowiak 85]. The implementation of the virtual machine implies that the operating system manages the physical resources of the machine (ex. memories, processors). We suppose that the physical architecture is *shared* between the multiple virtual machines dedicated to the various users' applications running in parallel. One of the main interests for sharing is to optimize the physical resource usage. For example, if a process performs a blocking i/o operation, the processor may be allocated to another process - belonging to the same application or to another application in order to increase the processor use rate. The notion of virtual machine is very adequate to resolve the competing accesses of the processes to the physical resources. To each resource, the operating system associates as many imaginary copies (virtual resource) as the number of processes competing for it.

Before dealing with distributed operating systems running on distributed architectures, we briefly recall some techniques applied in (centralized) operating systems running on nondistributed architectures. A fundamental scheduling technique on a uniprocessor machine is to apply *time-sharing*: a processor quantum is allocated to each process (virtual processor), the virtual processors are managed in a round order. Performances of such scheduling strategies have been evaluated, the reader can refer to [Muntz 75] for a synthesis. Scheduling on a multiprocessor equipped with a systemwide shared memory has also given rise to numerous works. In [Ousterhout 82], the notion of *coscheduling* is proposed so as to run simultaneously on different processors the runnable processes of an application thereby permitting highly communicating processes running at the same time. This follows from the fact that interprocessor communication takes place by the shared memory and thus is fast. Based on this assumption, coscheduling communicating tasks avoids the overhead of context switches that would occur if the tasks were not scheduled at the same time.

Scheduling in distributed operating systems (running on physically distributed architectures) is a difficult subject. In order to divide the complexity of the problem, we can attempt to identify two levels: the user-level scheduling and the system-level scheduling.

- *system-level scheduling.* One objective of this function is to operate a *fair* sharing of the processor resources between the users' applications that may run simultaneously. A second objective is to offer a set of basic mechanisms to allow the definition of various scheduling policies.

- *User-level scheduling.* This function has to schedule the processes of an application onto the processors allocated to it by the system scheduling level. The scheduling policy can be defined by the user himself according to the principles described in the previous sections. By

default, the operating system can also offer some scheduling policies as explained further.

Among the fundamental mechanisms offered by the system-level scheduling, let mention:

1. *Assignment.* The role of this mechanism is to assign a newly created process to a given processor allocated to the application; the choice policy of this processor may be user-defined or (by default) system-defined.

2. *Local scheduling.* This service (local to a processor) is similar to centralized operating system scheduling, it schedules the processes mapped onto the processor according to a scheduling policy which may be user-defined or provided (by default) by the operating system (ex. time-sharing). The allocation of the processor to a given process is performed by a basic mechanism of the operating system usually called the *dispatcher.*

3. *Migration.* This mechanism allows to move a process from its current machine onto another recipient machine. Note that such a mechanism has not to be used if a static mapping of the processes is performed (section 3).

The division between policies and mechanisms is a principle which is largely reflected in modern operating systems (ex. the Mach kernel [Black 90]). The separation between user-level and system-level scheduling is attractive since it allows a clear distinction between the processor allocation function and the use of these processors by the application itself. However, to our knowledge, no operating system does offer a strict separation between these levels, this is partly due to the difficulty to share the machine, with fairness, between the various applications which are submitted to the system according to a random distribution law.

As stated above, it appears useful that the system does offer a set of default scheduling policies so as to respond to situations where the applications do not wish to define specific policies by themselves. For example, it is desirable that the system schedules the processes belonging to an application when a user wishes to use the distributed system as a virtual centralized one without caring about the distribution. In that case, the 'user-level' scheduling has to be performed by the operating system itself. Note also that the implementation of user policies may rely on the availability of default system policies in order to simplify the work to be done at the user level [Black 90].

When applying a default scheduling policy, the majority of operating systems consider in first approximation that the set of processes belonging to the collection of running applications define a 'single' parallel application which exhibits dynamic parallelism as defined in section 2. Consequently, dynamic scheduling algorithms studied in section 4 are potentially applicable. In general, the default scheduling policy applied by operating systems, is to *balance the load* between the processors. However, most operating systems assume a coarse-grain parallelism of the running processes and consequently tend to privilege strategies which incur a low level of

migration. Mechanisms to measure the load are discussed in section 5.1. Load balancing can be performed either when a new process must be initially assigned to a processor or at some later stage applying a migration strategy. This is respectively discussed in sections 5.2 and 5.3. We conclude the section by some comments.

### 5.1 Load measuring

The load value of a processor must denote the processor capacity to execute new processes. Many of the load balancing algorithms use the processor usage rate to evaluate the load; yet others take the queue length of the processes in a ready state as a measure of a processor load. The load is balanced if the load of each processor is near the average. A processor is *overloaded* if its load is greater than the average and *underloaded* in the contrary. In order to compute the average load, it is necessary that (i) each processor computes its local load; and (ii) the processors exchange these values. The exchange periodicity may vary depending on the algorithms used; we do not detail such algorithms here, the interested reader can refer to [Tanenbaum 85, Barak 85b, Barak 85a, Barak 89].

### 5.2 Load balancing at initial assignment

If load balancing occurs when (and only when) the initial assignment of a process has to be determined, we say that *load balancing at initial assignment* is performed. In other words, migration is prohibited [1]. Should a new process be created on a site called the *origin site*, the algorithm looks for an adequate machine and allocates this processor to the process. Recall from section 4 that a bidding algorithm is adequate for that purpose. There exists several ways to implement such a bidding method. In most cases, the difference lies in the implementation of the first step of the algorithm (sending requests containing the characteristics of the process to execute). Some algorithms send (broadcast) requests to all other sites of the system [Tanenbaum 85]. Others broadcast requests to a restricted set of neighbouring machines; if no machine answers positively, broadcasting is widened [Boillat 90]. Finally, some algorithms do only enquire after the inactive sites which are supposed to be known [Rost 90, Cova 88].

### 5.3 Load balancing by migration

When dynamic load balancing by migration is performed, a process may migrate during its execution. Recall that migration implies a decision policy as well as a transfer mechanism. We examine each of these aspects in the following.

[1] This type of scheduling algorithms is sometimes called static load balancing; we will not use this term here to avoid the confusion that could be made with the static mapping algorithms described in section 3.

### 5.3.1 Decision policy

In an operating system, many events may be the cause of a migration. They fall into two categories. The first entails the events related to the load while the second entails the global characteristics of a process (observed during execution) on a site. Other factors could lead to migration. For instance, when a machine is stopped for maintenance purposes, processes residing on that machine may be transferred onto another site. However, such events are not considered in the following.

**Load decisions.** Load examination can be performed at various stages. Two situations at least are to be considered. The first concerns an inactive site. A second situation is to periodically compute the average load of the system. Possible migration of processes from overloaded machines to underloaded machines can then be envisaged.

**Decisions depending on processes' behaviour.** Recall that in this section, we are assuming that no a priori informations about the processes' characteristics (ex. execution time required) are known statically. However, a dynamic load balancing algorithm may accumulate some information about a process behaviour (during the execution) and attempt to take it into account in a migration decision so as to optimize performances. An essential parameter to be considered for instance is the communication pattern of a process. If a process does communicate intensively with another process residing on a remote site or accesses a lot of objects residing on another site, it might be more appropriate to execute the process on the remote site.

**Prevention of excessive migrations.** Assuming a coarse-grain of parallelism (heavy processes), migration is a costly operation, therefore an excessive number of migrations may deteriorate the overall performances. Several heuristics are used to limit the migrations [Barak 85b]. We introduce two such heuristics below:

- The load difference between two machines involved in a migration must be sufficiently high to justify the migration of processes from the most loaded machine to the least loaded one.
- Before being candidate to migration, a process must consume a given number of processor quantum on the current site. This prevents short processes from supporting the costs of a migration.

Notice that the various decision parameters introduced previously could be conveniently modelled in the framework of Stankovic's algorithm described in section 4. Load consideration decisions and process behaviour characteristics would act as stimuli while prevention of excessive migrations heuristics would act as inhibitors.

### 5.3.2 Transfer mechanism

A process transfer can be negotiated either by the current machine or the recipient machine of the migrating process. A transfer algorithm moves a process from its current machine to the recipient machine in three steps [Lu 89]:

1. The process is suspended on the current site.
2. The process context is moved onto the recipient machine.
3. The migrating process is reactivated on the recipient machine.

The process context may contain several informations: (i) the processor context (machine status, registers); (ii) current positions in opened files and values of the timers; and (iii) the process address space in memory.

There exists several ways to transfer the address space [Ousterhout 87]. Below, we present some examples borrowed from existing distributed operating systems which support migration.

In the systems Demos/mp [Powell 83] and Locus [Walker 83], a migrating process is suspended during the whole transfer of its address space to the recipient machine.

The system System-V [Cheriton 84] performs a precopy of the address space so as to decrease the time during which the migrating process is suspended. The address space is sent to the recipient machine while the process pursues its execution on the current machine. Of course, the process may modify some pages which have already been sent. These pages are sent again per blocks to the recipient machine. When the number of remaining dirty pages is sufficiently small, the process is suspended; the environment together with the remaining dirty pages is then transmitted to the recipient.

The Accent [Rashid 81] system takes advantage of the *single level store memory* provided by the system to facilitate the transfer. The (virtual) address space is demand paged. When a transfer is required, only the process page table is transferred. When the process restarts its execution on the recipient machine, pages will be transferred on demand according to the process references in virtual memory. Using the *copy-on-reference* approach, the Accent system avoids to move the entire address space of a migrating process and thereby reduces significantly the migration cost.

Besides the issues above, an operating system allowing migration, must be capable to deliver corrrectly the messages to processes whose location has changed. The reference [Lu 89] describes several techniques that may be used by the communication subsystem.

## 5.4 Comments

The default scheduling policy used in the majority of operating systems is to balance the load either at initial assignment time of a new process or more dynamically by migration during

of their computation to the parent. The properties of the execution graph induced by the model of computation have been genuinely used by the designers of the scheduling algorithm explained above. For instance, the possible migration of a task during execution for communication optimization reasons is not of concern in this example given the properties of the communication patterns induced by the model of computation.

## 6.2 Object oriented machines

Object oriented programming leads to the decomposition of an application into abstraction and protection units called *objects*. An object encapsulates some private data called - instance variables - and exported procedures (operations) called *methods*. Some object oriented languages allow inheritance of properties between classes. We do not detail more these aspects here, the interested reader may refer to [Masini 89] for instance.

The introduction of parallelism into object oriented languages has given rise to numerous propositions [Yonezawa 87]. In the following, we examine the principles of the execution machine of the Pool language [America 87]. Pool adopts a model of *active* object. Each object, called (in a synonymous way) a process contains a *body* of imperative sequential instructions which command the object behaviour during its life. Processes can be dynamically created during the program execution and communicate via message passing.

In order to explain the implementation principles of the language execution model, the authors [America 87] first introduce the notion of an *abstract machine* consisting of an 'infinite' number of *sites* communicating through a virtually fully connected network. Each site is equipped with a processor, a local memory and a communication unit; a site is in charge of the execution of a *single* object. The execution context of an object contains: (i) a stack which size may dynamically evolve; (ii) a First In First Out queue memorizing the requests addressed to the object; and (iii) a block of code.

Of course, such a machine would be too much costly if not impossible to build at the architecture level. The real machine designed to execute Pool programs is called Doom ("Decentralized Object Oriented Machine"). In first approximation, this machine is a reduction of the previous abstract machine to a *finite* number of sites which are managed by the operating system. Several objects may share a same physical site.

We describe now the object creation process. Pool hides the distribution; however, a Pool program can be annotated by pragmas which indicate on which potential sites an object could be created. Notice that in such a case, it is the programmer himself who acts (explicitly) on the scheduling decisions. When object creation takes place, the operating system dynamically selects (among the potential sites indicated by the programmer) a site which leads to an optimization of resource usage. The optimization is guided by load balancing criteria, memory contention and communication rate over the network.

The virtual machine adapted to the language execution model is implemented by the operating system. In other words, there is an adequacy between the language oriented virtual machine and the operating system which manages the resources of the distributed system. The scheduling strategy applied (at the moment this report is written) boils down to a dynamic scheduling algorithm similar to an initial assignment algorithm described in sections 4 and 5. While this strategy may lead to good results in some environments, we may question whether it would be the case in all object environments. In our view, this may depend, for instance, on the object granularity and the object creation rate.

The influence of the computation model on the scheduling algorithm proposed by the Doom designers is, in this case, undoubtedly less obvious than in the previous example. However, we can notice that the computation model allows for the dynamic creation of objects and therefore makes some dynamic scheduling mandatory.

# 7 Conclusions and questions

Scheduling parallel applications on a distributed architecture is a complex subject about which numerous research works have been undertaken. The overall goal of scheduling a parallel application, as studied in this report, is to reduce the application response time. Two subproblems have to be solved: (i) the mapping of the parallel units to the processors, and (ii) the local scheduling on each processor. The main factors that impinge on the application performances are: (i) the processors's load, and (ii) the communication costs.

When an application exhibits static parallelism and the architecture configuration is not subject to changes, the mapping of the parallel units can be tackled prior to execution. However, this requires that some basic characteristics (execution time, communication patterns) are estimated statically. These conditions severely limit the applicability of static policies in practice.

In contrast to static methods, dynamic scheduling algorithms are applied during the execution itself so as to adapt the mapping to the dynamically changing situation. Most of dynamic methods model the scheduling problem by an objective function which is optimized several times during the execution for taking into account the dynamic aspects of the computation.

It should be noted that a static method may be usefully combined with a dynamic algorithm; the result of the static work being the starting point of the execution. In sections 3 and 4, we have introduced a set of static and dynamic scheduling methods. Clearly, not all these methods equally apply to the various scheduling contexts. For instance, the physical model introduced in section 4.3 would appear much more adequate in a very dynamic context where numerous fine-grained processes are to be scheduled on a massively parallel architecture rather than in a general purpose operating system where a limited number of coarse-grained processes are competing for the processor resource.

This comment appeals for a more general research work which would consist in classifying these various algorithms according to relevant parameters. The performances of the algorithms could then be evaluated under various values of those parameters so as to give some hints to designers for choosing tools and methods adequate to their problems.

As stated in section 2, we have been primarily interested in scheduling logically distributed software where communication takes place through message passing and we have paid little attention to models of computation where communication takes place through shared data. A research perspective would be to assess to which extent scheduling methods based on a message passing model of communication could also be applied when communication takes place through shared data.

Scheduling in general purpose operating system, discussed in section 5, has to take into account a complementary problem namely the sharing of the machine between multiple applications which may be submitted to execution according to a random distribution. Two scheduling levels are then distinguished: (i) the user-level scheduling; and (ii) the system-level scheduling. The role of the system-level scheduling is on the one hand, to realize a fair sharing of the processor resource between the various applications possibly pertaining to different users and on the other hand, to provide the basic mechanisms in order to implement various scheduling policies which may be user-defined or (by default) system-defined.

Static scheduling algorithms incur a low overhead during execution and have the merit to define a rigorous framework for performance evaluation. However, as stated before, they are limited to applications which characteristics must be exhibited statically; they do not take into account either the sharing of the architecture between various applications. Dynamic scheduling algorithms relax the constraint that application characteristics must be exhibited statically at the price of a higher overhead during execution but still do not address the sharing problem. General purpose operating systems do address the sharing of the machine between several applications but, given the diversity of the applications submitted to execution, make little use of the application characteristics in order to optimize the performances. Yet, application characteristics are taken into account only in a dynamic fashion.

These remarks raise the following question: would it be possible to design a scheduling scheme allowing (i) to integrate both static and dynamic scheduling of an application; and (ii) to share the machine between multiple applications? In our view, the notion of *virtual machine* would appear an adequate framework for solving this problem as discussed in the following.

A virtual machine would be dedicated to each application; the architecture being shared between multiple virtual machines in order to allow for the parallel execution of various applications. A virtual machine should take advantage of the application characteristics so as to execute the application efficiently. How could this be achieved is briefly sketched in the following. First, the virtual machine should offer a *judicious interface* so as to facilitate the static scheduling (ba-

sed on the characteristics exhibited statically) of an application. A static algorithm similar to those described in section 3 would map the application processes onto the interface of the virtual machine. Second, the virtual machine should be carefully implemented so that the scheduling decisions taken at static scheduling time are not invalidated during the execution. Rather, the implementation of the virtual machine should ideally complement the static decisions taking into account the dynamic characteristics of the applications.

Given the complexity of the scheduling problem, it might be worth, in practice, experimenting and evaluating various static scheduling policies as well as different implementation algorithms. To this end, we can imagine to develop *multiple virtual machines*. Notice that the class inheritance facility of an object oriented language might be very useful to that purpose as exemplified, for instance, by the Choices object-oriented system [Campbell 89].

Satisfying the objectives of such a virtual machine in full generality is clearly very hard but given the particular model of computation of a language oriented machine, it is our belief that the problem is less difficult to solve since the knowledge of some logical properties characterizing the applications written in the language can facilitate the scheduling work. For example, in an object machine, the execution of an object's method is statistically a short duration process; therefore, it might be unnecessary to envisage the migration of such a process during its life. Yet, it is usually the case that the processor allocation function of a virtual machine interferes with other resource allocators such as memory management, as far as performances are concerned. Again, taking into account the interference between the various resource allocators appears easier within the framework of language oriented machines.

As a final comment, it is worth noting that the scheduling algorithms of the language oriented machines described in section 6 implement quite rough strategies in light of the previous discussion. For instance, the virtual machine Doom introduced by the designers of the Pool language (section 6.2) is distinct in many respects from the proposal above. First, in the proposal above, a virtual machine is dedicated to each application while we understand that Doom offers a single virtual machine for the entire set of parallel applications. Second, the interface of the Doom machine, although very simple, is inappropriate for static scheduling: there are no informations available at the interface which would allow a static scheduling algorithm to cluster/scatter objects so as to take advantage of the (statically exhibited) characteristics of an application. This justifies, in our view, further research in this area; such a work is under way for the design of a parallel object oriented language [Benveniste ] within the Gothic [Banâtre 91] project.

## Acknowledgements

It is a pleasure to gratefully acknowledge the valuable comments made on earlier drafts of this paper by some of our colleagues at Irisa: M. Banâtre, V. Issarny, Y. Jégou, T. Leconte and D.

Le Métayer. This work has been partly funded by the Ministery of Education of Algeria.

## References

[America 87] P. America. POOL-T : A Parallel Object-Oriented Language. In *Object-Oriented Concurrent Programming*, pp 199–220. MIT Press Series in Computer Systems, 1987.

[Andre 88] F. Andre and J-L. Pazat. Le Placement de Tâches sur les Architectures Parallèles. *Techniques et Sciences Informatiques*, 7(4):385–401, 1988.

[Anger 90] F.D. Anger, J. Hwang, and Y. Chow. Scheduling with Sufficient Loosely Coupled Processors. *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

[Bal 89] H.E. Bal, G. Steiner, and S. Tanenbaum. Programming Languages for Distributed Computing Systems. *Computing Surveys*, 21(3), September 1989.

[Banâtre 90] J.P. Banâtre. *La Programmation Parallèle : outils, méthodes et éléments de mise en œuvre*. Eyrolles, 1990.

[Banâtre 91] J.P. Banâtre and M. Banâtre, editors. *Les systèmes distribués : l'expérience du système Gothic*. InterEditions, February 1991.

[Barak 85a] A. Barak, N. Alon, and U. Manber. On Disseminating Information Reliably without Broadcasting. *Computer Sciences Technical Report*, (621), December 1985.

[Barak 85b] A. Barak and A. Shiloh. A Distributed Load-Balancing Policy for a Multicomputer. *Software Practice and Experience*, 15(9):901–913, September 1985.

[Barak 89] A. Barak and R. Wheeler. MOSIX : An Integrated Multiprocessor UNIX. In *Proc. of the Winter 1989 USENIX Conference*, pp 101–112, San Diego, CA, February 1989.

[Benveniste ] M. Benveniste and V. Issarny. Arche : un Langage Parallèle à Objets. Rapport de recherche (to appear), INRIA.

[Black 90] D.L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. Technical Report CMU-CS-90-125, Department of Computer Science, Carnegie-Mellon University, April 1990.

[Boillat 90] J.E. Boillat and P.G. Kropf. A Fast Distributed Mapping Algorithm. *LNCS CONPAR 90-VAPP IV*, pp 405–416, September 1990.

[Bokhari 81] S.H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30(3):207–214, March 1981.

[Campbell 89] R. H. Campbell, G. M. Johnston, P. W. Madany, and V. N. Russo. Principles of Object-Oriented Operating System Design. Technical Report R-89-1510, University of Illinois at Urbana-Champaign, April 1989.

[Cheriton 84] D.R. Cheriton. The V-Kernel: A Software Base for Distributed Systems. *IEEE Transactions on Software Engineering*, 1(2):19–43, 1984.

[Chrétienne 91] P. Chrétienne and F. Lamour. Mapping Graphs onto a Partially Reconfigurable Architecture. In A. Bode, editor, *Distributed Memory Computing*. Springer Verlag, April 1991.

[Chu 80] W.W. Chu, L.J. Holloway, M.T. Lan, and K. Efe. Task Allocation in Distributed Data Processing. *IEEE Computer*, pp 57–69, November 1980.

[Colin 89] J-Y. Colin. *Problèmes d'Ordonnancement avec Délais de Communication : Complexité et Algorithmes*. PhD thesis, Univ. de Paris VI, November 1989.

[Cova 88] L.L Cova and R. Alonso. Distributing Workload Among Independently Owned Processors. Technical Report CS-TR-200-88, Princeton university Department of computer sciences, December 1988.

[Ercal 90] F. Ercal, P. Sadayappan, and J. Ramanujam. Task Allocation onto a Hypercube by Recursive Mincut Bipartitionning. *Journal of Parallel and Distributed Computing*, (10):35–44, 1990.

[Ferrari 87] D. Ferrari and S. Zhou. An Empirical Investigation of Load Indices for Load Balancing Applications. In P.J. Courtois and G. Latouche, editors, *Performance87*. North-Holland, 1987.

[Fox 88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, Salmon. J., and D. Walker. *Solving Problems on Concurrent Processors*. Prentice-Hall International, 1988.

[Fradet 89] P. Fradet and D. Le Métayer. Compilation of Functional Languages by Program Transformation. Technical Report 1040, INRIA, May 1989.

[Giorgi 90] J-F Giorgi and D. Le Métayer. Continuation-Based Parallel Implementation of Functional Languages. In *ACM Conf. on Lisp and Functional Programming*, July 1990.

[Ji 90] J. Ji and M. Jeng. Dynamic Task Allocation on Shared Memory Multiprocessor Systems. In *Proc. of International Conference on Parallel Processing*, pp 17–21, 1990.

[Kim 88] S.J. Kim. *A General Approach to Multiprocessor Scheduling*. PhD thesis, Department of computer sciences, Texas university, February 1988.

[Kirimis 88] K. Kirimis and R. Alonso. An Experimental Comparison of Initial Placement vs. Process Migration for Load Balancing Strategies. Technical Report CS-TR-199-88, Princeton university Department of computer sciences, December 1988.

[Krakowiak 85] S. Krakowiak. *Principes Des Systèmes d'Exploitation Des Ordinateurs.* Dunod, 1985.

[Lo 85] V.M. Lo. Task Assignment to Minimize Completion Time. In *Proc. of 5th International Conference on Distributed Computing Systems*, pp 239–336, 1985.

[Lo 87] V.M. Lo. Heuristic Algorithm for Task Allocation in Distributed Computer Systems. Technical Report CIS-TR-86-13, Department of computer sciences University of Oregon, April 1987.

[Lu 89] C. Lu. *Process Migration in Distributed Systems.* PhD thesis, University of illinois, January 1989.

[Manber 89] U. Manber. *Introduction to Algorithms: a Creative Approach.* Addison wesley, 1989.

[Masini 89] G. Masini, A. Napoli, D. Leonard, and K. Tombre. *les Langages à Objets.* Inter-Editions, 1989.

[Muntz 75] R.R. Muntz. Scheduling and Resource Allocation in Computer Systems. *Sciences Research Associates*, pp 269–307, 1975.

[Ousterhout 82] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems.* IEEE, 1982.

[Ousterhout 87] J. Ousterhout and F. Douglis. Process Migration in the Sprite System. Technical Report CA 94720, Computer Science Division, University of California, Berkeley, September 1987.

[Powell 83] M.L. Powell and B.P. Milner. Process Migration in DEMOS/MP. In *Proc. of 9th ACM Symposium on Operating Systems Principles*, pp 110–119, October 1983.

[Ramanritham 89] K. Ramanritham and J.A Stankovic. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers*, 38(8):49–59, August 1989.

[Rashid 81] R.F. Rashid and G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proc. of 8th ACM Symposium on Operating Systems Principles*, pp 64–75, Pacific Grove, December 1981.

[Rost 90] J.C. Rost and J.S.K. Wong. A Distributed Algorithm for Dynamic Scheduling. *LNCS CONPAR 90-VAPP IV*, pp 628–639, September 1990.

[Ryou 88] J.C. Ryou and J.S.K. Wong. A Heuristic Algorithm for Task Allocation in Distributed Computer Systems. Technical Report TR 88-7, Department of computer sciences IOWA state university, March 1988.

[Sadayappan 90] P. Sadayappan, F. Ercal, and J. Ramanujam. Cluster Partitionning Approaches to Mapping Parallel Programs onto Hypercube. *Parallel Computing*, (13):1–16, 1990.

[Schwan 85] K. Schwan and G. Gaymon. Automatic Allocation for Cm Multiprocessor. In *Proc. of 5th International Conference on Distributed Computing Systems*, pp 310–320, May 1985.

[Stankovic 84] J.A. Stankovic and I.S. Sidhu. An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups. *IEEE Transactions on Software Engineering*, 15(11):49–59, 1984.

[Stone 77] H.S. Stone. Multiprocessor Scheduling with Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[Tanenbaum 85] A.S. Tanenbaum and R.V. Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17(4), December 1985.

[Ullman 75] A.S. Ullman. NP-Complete Scheduling Problem. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[Walker 83] B. Walker and al. The LOCUS Distributed Operating System. In *Proc. of 9th ACM Symposium on Operating Systems Principles*, pp 49–70, October 1983.

[Yonezawa 87] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press Series in Computer Systems, 1987.

## LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 584 TECHNIQUES POUR LA MISE AU POINT DE PROGRAMMES REPARTIS
Michel ADAM, Michel HURFIN, Michel RAYNAL, Noël PLOUZEAU
Mai 1991, 10 Pages.

PI 585 TOWARDS THE CONSTRUCTION OF DISTRIBUTED DETECTION PROGRAMS, WITH AN APPLICATION TO DISTRIBUTED TERMINATION
Jean-Michel HELARY
Mai 1991, 24 pages.

PI 586 OPAC : A COST-EFFECTIVE FOATING-POINT COPROCESSOR
André SEZNEC, Karl COURTEL
Mai 1991, 26 Pages.

PI 587 ON FAILURE DETECTION AND IDENTIFICATION : AN OPTIMUM ROBUST MIN-MAX APPROACH
Elias WAHNON, Albert BENVENISTE
Mai 1991, 24 Pages.

PI 588 BOUNDED-MEMORY ALGORITHMS FOR VERIFICATION ON THE FLY
Claude JARD, Thierry JERON
Mai 1991, 14 pages.

PI 589 UNE APPROCHE MULTIECHELLE A L'ANALYSE D'IMAGES PAR CHAMPS MARKOVIENS
Patrick PEREZ, Fabrice HEITZ
Juin 1991, 32 pages.

PI 590 THE IDEMPOTENT SOLUTIONS OF THE SEMI-UNIFICATION PROBLEM
Pascal BRISSET, Olivier RIDOUX
Juin 1991, 16 pages.

PI 591 AVARE UN PROGRAMME DE CALCUL DES ASSOCIATIONS ENTRE VARIABLES RELATIONNELLES
Mohamed OUALI ALLAH
Juin 1991, 32 pages.

PI 592 SCHEDULING IN DISTRIBUTED SYSTEMS : SURVEY AND QUESTIONS
Yasmina BELHAMISSI, Maurice JEGADO
Juin 1991, 36 pages.



Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399